Relatório do Trabalho Prático de POO: Sistema de Gestão de Obra de Construção Civil

Bruno Paiva
a31496@alunos.ipca.pt

IPCA - Licenciatura de Engenharia de Sistemas Informáticos

Novembro 2025

Resumo

Este relatório descreve o desenvolvimento do trabalho prático da Unidade Curricular de Programação Orientada a Objetos (POO), focado no tema **Gestão de Obra de Construção Civil**. O objetivo principal é a aplicação do Paradigma Orientado a Objetos para modelar e gerir os custos associados a uma obra, incluindo materiais, armazéns, mão de obra (própria e subcontratada), veículos, e orçamentos. O documento detalha a estrutura de classes, a arquitetura da solução em múltiplas camadas (`.Core`), o uso de interfaces para promover o desacoplamento e a aplicação de exceções customizadas para um tratamento de erros robusto. O projeto demonstra a consolidação de conceitos POO e boas práticas de desenvolvimento em C#, cumprindo os requisitos iniciais para a Fase 1.

Conteúdo

Capítulo 1

Introdução

1.1 Motivação e Tema

O trabalho foi desenvolvido no contexto da UC de POO, com o propósito de aplicar conceitos POO na resolução de problemas reais de complexidade moderada. O tema escolhido, **Gestão de Obra de Construção Civil**, envolve o controlo e a gestão dos custos de uma obra específica. As entidades-chave modeladas incluem: materiais, armazéns, stocks, viaturas, e mão de obra/serviços.

1.2 Objetivos da Solução

Os principais objetivos de desenvolvimento, alinhados com os requisitos da Fase 1, foram:

- Analisar o problema e identificar a estrutura de classes e estruturas de dados.
- Desenvolver a solução em C#, aplicando interfaces, herança, abstração e encapsulamento.
- Implementar o cálculo dos custos totais da obra e a variação orçamental.
- Utilizar exceções customizadas para um tratamento de erro limpo e específico.
- Estruturar o projeto em camadas (DLLs) para modularidade e reutilização.

Capítulo 2

Arquitetura e Estrutura da Solução

2.1 Arquitetura em Camadas

A solução foi arquitetada em duas camadas principais, utilizando o conceito de bibliotecas (.DLL) para separação de responsabilidades, conforme as boas práticas de programação por camadas:

- **Projeto_POO.Core**: Contém toda a **lógica de negócio** (Classes e Interfaces). Esta biblioteca define as entidades e as regras de gestão da obra, sendo independente de qualquer interface de utilizador, promovendo reutilização e extensibilidade.
- **Projeto_POO**: Contém o programa principal (`Program.cs`) e a **lógica de apresentação** e interação com o Core (atualmente, a aplicação demonstradora).

2.2 Hierarquia de Classes e Interfaces

A estrutura central do sistema é baseada na classe agregadora `ConstructionWork`. A solução utiliza um conjunto de interfaces para definir contratos e tipificar as diferentes entidades, promovendo o desacoplamento e a aplicação do Polimorfismo.

2.2.1 Interfaces Fundamentais

As interfaces definem as capacidades essenciais de cada entidade, assegurando a conformidade e a extensibilidade do sistema:

- `ICostable`: Contrato para todas as entidades que contribuem para o custo total da obra (`GetCost()`).
- `IDescribable`: Garante que a entidade possui uma descrição ou nome (`GetDescription()`).
- `IReportable`: Contrato para a geração de relatórios formatados (`GetReport()`).
- `IIdentifiable`: Define uma identificação única para a entidade (`GetId()`).
- `IDateable`: Garante que a entidade possui uma data associada.
- `IStorable`: Define capacidades de gestão de stock (quantidade, adicionar, remover).

2.2.2 Classes Abstratas e Concretas

A arquitetura do sistema é construída em torno da seguinte hierarquia e relações:

CostableItem (Classe Abstrata)

`CostableItem` é uma classe abstrata que serve como base para todas as entidades que possuem um custo e uma descrição. Ela implementa as interfaces `ICostable` e `IDescribable`, fornecendo uma implementação base para os métodos `GetCost()` e `GetDescription()`.

Labor (Mão de Obra / Serviços)

`Labor` herda de `CostableItem`. Esta classe representa tanto a mão de obra própria quanto os serviços subcontratados. Possui atributos como ID, nome, custo por hora, horas trabalhadas e um flag `_isSubcontracted` para diferenciar os tipos de mão de obra/serviços. Implementa também `IIdentifiable`, `IDateable` e `IReportable`.

Material

A classe `Material` representa os materiais utilizados na obra, com atributos como ID, nome, unidade de medida e preço unitário. Ela implementa `ICostable`, `IDescribable` e `IIdentifiable`.

Storage (Armazém)

`Storage` representa um armazém onde os materiais são guardados. Contém uma lista de `StorageItems`. Implementa `IDescribable`, `IIdentifiable` e `IReportable`.

StorageItem (Item em Armazém)

Um `StorageItem` representa uma entrada específica de um `Material` dentro de um `Storage`, incluindo a quantidade desse material. Este item calcula o seu custo com base na quantidade e no preço do `Material` associado. Implementa `ICostable`, `IStorable` e `IReportable`.

Vehicle (Veículo)

A classe `Vehicle` representa uma viatura, com atributos como ID, matrícula e custo por hora. Implementa `IDescribable`, `IIdentifiable` e `IReportable`.

VehicleUsage (Uso de Veículo)

`VehicleUsage` regista a utilização de um `Vehicle` específico, registando as horas de uso. Calcula o custo total com base nas horas e no custo por hora do `Vehicle` associado. Implementa `ICostable` e `IReportable`.

Budget (Orçamento)

A classe `Budget` representa uma rubrica orçamental, com um nome e um valor definido. Implementa `ICostable` e `IDescribable`.

Document (Documento)

`Document` representa um documento associado à obra, com ID, tipo, data e descrição. Implementa `IDescribable`, `IIdentifiable`, `IDateable` e `IReportable`.

ConstructionWork (Obra de Construção)

Esta é a classe central e agregadora do sistema. `ConstructionWork` contém listas de `Storage`, `Labor`, `Vehicle`, `VehicleUsage`, `Budget` e `Document`. É responsável por agregar e calcular o custo total da obra, somando os custos de todos os itens agregados de forma polimórfica (através da interface `ICostable`).

2.2.3 Relações Principais

O sistema estabelece as seguintes relações entre as classes:

- **Herança (`CostableItem` → `Labor`)**: `Labor` herda de `CostableItem`, reutilizando a lógica base de custo e descrição.
- **Composição/Associação (`Storage` → `StorageItem` → `Material`)**: Um `Storage` contém múltiplos `StorageItem`s, e cada `StorageItem` referencia um `Material` específico.
- **Composição/Associação (`Vehicle` → `VehicleUsage`)**: Um `VehicleUsage` está associado a um `Vehicle` específico para registar o seu uso.
- **Agregação (`ConstructionWork` → Outras Classes)**: A classe `ConstructionWork` agrega (contém listas de) instâncias de `Storage`, `Labor`, `Vehicle`, `VehicleUsage`, `Budget` e `Document`, formando a estrutura completa de uma obra.
- **Realização de Interfaces**: Diversas classes realizam interfaces como `ICostable`, `IDescribable`, `IIdentifiable`, `IReportable`, `IDateable` e `IStorable`, garantindo a conformidade e a capacidade de interagir de forma polimórfica.

Capítulo 3

Aplicação dos Pilares POO

3.1 Encapsulamento

Todas as classes utilizam campos privados (`_field`) e propriedades ou métodos `Get/Set` públicos para controlar o acesso e a modificação dos dados, cumprindo o pilar de Encapsulamento. Isto protege o estado interno dos objetos.

3.2 Abstração e Polimorfismo

- A interface `ICostable` representa a **Abstração** de que "algo tem um custo".
- O método `totalCost()` em `ConstructionWork` demonstra **Polimorfismo** ao somar os custos de diferentes tipos de objetos (materiais, mão de obra, veículos, orçamentos) apenas chamando o método `GetCost()` definido na interface, sem se preocupar com a implementação interna de cada objeto.

3.3 Exceções Customizadas

Foram criadas exceções customizadas (estendendo `Exception`) para lidar com condições de erro específicas do domínio:

- `InvalidQuantityException`: Lançada ao tentar adicionar ou remover stock com quantidade ≤ 0.
- `MaterialNotFoundException`: Lançada ao tentar remover stock de um material inexistente.

- `InsufficientStockException`: Gerencia a falha na remoção quando o stock disponível é menor que o solicitado.

Capítulo 4

Conclusões e Trabalho Futuro

4.1 Conclusões

O projeto na sua Fase 1 cumpriu os requisitos de modelação e implementação essencial das classes, com uma clara aplicação dos pilares POO. A utilização de interfaces e classes abstratas promoveu o desacoplamento e a reutilização do código. A modelação da classe `Labor` para incluir serviços subcontratados simplificou a estrutura de custos, mantendo a coerência do sistema. A arquitetura em camadas e o uso de exceções customizadas estabelecem uma base robusta para a continuidade do desenvolvimento.

4.2 Trabalho Futuro (Fase 2)

Para a próxima fase (19-12-2025), os próximos passos incluem:

- **Implementação Final:** Refinar a lógica de negócio e as validações, corrigindo inconsistências como as referências remanescentes a 'Service' em `ConstructionWork.cs`.
- **Aplicação Demonstradora:** Implementar a aplicação demonstradora no projeto `Projeto_POO` para exercer os serviços implementados no `.Core`.
- **Testes Unitários:** Implementar testes para garantir uma cobertura mínima de 50% do código.
- **Persistência de Dados:** Adicionar a funcionalidade de guardar e carregar a obra em ficheiros.

4.3 Link do Repositório GitHub

Poderá aceder ao código-fonte completo através do seguinte link:
`https://github.com/yesisr/TP_POO_Fase-1`